2025
BERLIN

24 - 26 September, 2025 / Berlin, Germany

EVADING IN PLAIN SIGHT: HOW ADVERSARIES BEAT USER-MODE PROTECTION ENGINES FOR OVER A DECADE

Omri Misgav

Independent researcher, Israel

ABSTRACT

Over the years, user-mode protection engines have become an integral part of many endpoint-oriented security solutions, from malware analysis tools to commercial products (like NGAVs, EDRs and sandboxes), so it was only natural for attackers to develop various ways to get around them.

I have examined open-source projects and publications, and reverse-engineered malware families of all types in the wild, including infamous threats like Emotet, Ursnif (Gozi), SmokeLoader, FormBook, TrickBot, Lumma stealer and Winnti, among others.

The result is a comprehensive mapping of the entire threat landscape, along with in-depth insights into attackers' tradecraft and how adversaries have evolved over more than a decade.

This paper documents the unique methods devised by security researchers and malware authors to beat user mode-based protection engines. In addition to a complete taxonomy, it also outlines a methodology for both real-time and forensics detection that is suitable for manual procedures as well as integration into enterprise-grade solutions.

This work aims to serve as a definitive guide on this subject for security researchers, malware analysts, threat hunters, incident responders and detection engineers.

INTRODUCTION

To beat endpoint-oriented security solutions attackers try to either break apart execution chains to obscure context or disrupt visibility by blinding them to the operations executed by a process.

A user-mode protection engine operates within Ring 3 (user space) of the operating system, via instrumentation (e.g. AMSI [1]) or hooking. The engine gathers telemetry and implements the blocking mechanisms upon detection of malicious activity, it isn't necessarily the component making the detection decisions.

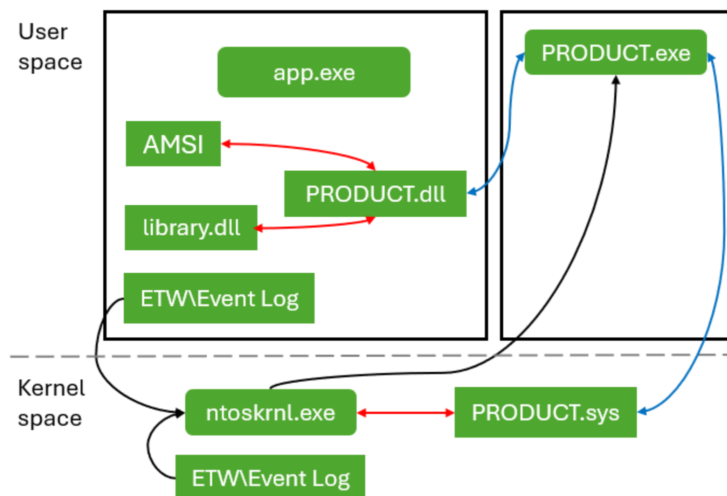

Figure 1: High-level architecture of endpoint-oriented security product.

The research focuses on the *Windows* operating system, where user-mode protection engines are most commonly deployed.

Technical background

Hooking is used to alter or augment a function's behaviour by intercepting calls to it. User-mode hooks are used for a number of reasons:

1. Simplicity and stability

   User-mode code is generally easier and safer to develop as it runs with less privilege, posing a lower risk to affect the entire system. Nevertheless, it can still introduce instability, particularly due to 'Critical Process' [2].

2. Patch protection

   The kernel of 64-bit versions of *Windows* is equipped with Kernel Patch Protection (KPP or PatchGuard), which detects modifications to its code and data structures. *Windows 10* has also implemented HyperVisor Code Integrity (HVCI [3]), which prevents modification to its code in the kernel.

3. Full context

   Certain information is only accessible in user-mode. For instance, when web browsers use SSL/TLS connections the content (such as URL, headers and data) is only visible in user space to the application that encrypts and decrypts it.

Linux doesn't offer patch protection out-of-the-box, making it possible to apply hooks directly in the kernel for system-wide coverage. In *macOS*, System Extensions (which power Endpoint Security Framework [4] and Network Extensions [5]) don't have direct access to the kernel, thus kernel hooks cannot be applied and they cannot be used to inject user-mode components into processes.

There are different layers and locations where hooks can be placed to control a specific desired functionality of the system, like ANSI\Unicode versions of functions in kernel32.dll, internal system DLLs like kernelbase.dll or ntdll.dll, and the WOW64 emulation layer DLLs.

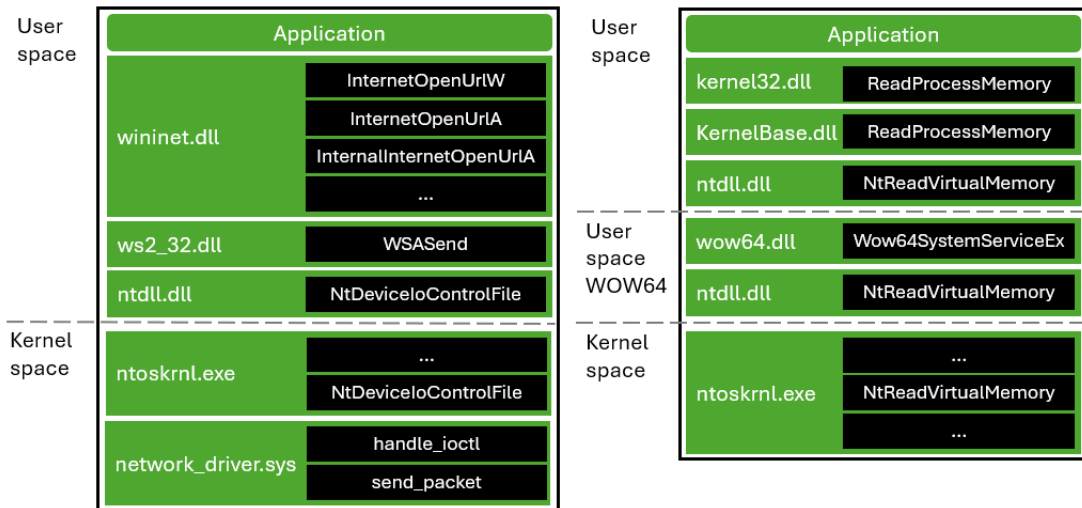

Figure 2: Illustration of system service call stacks.

There are two primary methods to hook a function: inline hooking and Import Address Table (IAT) hooking.

Inline hooks change the code of the target function in-memory, usually using control flow instructions (`jmp` \ `call` \ `push` + `ret`).

```
function_A:
0x801000: 55                push ebp
0x801001: 89 e5             mov  ebp, esp
0x801003: 83 ec 40          sub  esp, 0x40
0x801006: 50                push eax
0x801007: 8b 44 24 0c       mov  eax, [esp+0xc]
0x80100a: ...
```

```
function_A:
0x801000: e9 95 09 00 00    jmp  hook_A
0x801005: 90                nop
0x801006: 50                push eax
0x801007: 8b 44 24 0c       mov  eax, [esp+0xc]
0x80100a: ...

hook_A:
0x802000: 55                push ebp
0x802001: 89 e5             mov  ebp, esp
0x802003: 83 ec 40          sub  esp, 0x40
0x802006: e9 fc ef ff ff    jmp  function_A+0x6
```

Figure 3: Illustration of inline hook using control flow instruction.

Another way to hook a function is by having it generate exceptions, either with breakpoints or invalid opcodes, but that requires being able to catch the exceptions. Debuggers implement this approach, typically by applying software breakpoints (`int 3` instruction) in memory as the number of hardware breakpoints is very small.

```
function_A:
0x801000: 55          push ebp
0x801001: 89 e5       mov  ebp, esp
0x801003: 83 ec 40    sub  esp, 0x40
0x801006: ...
```

```
function_A:
0x801000: cc          int  3
0x801001: 89 e5       mov  ebp, esp
0x801003: 83 ec 40    sub  esp, 0x40
0x801006: ...
```

Figure 4: Illustration of an applied software breakpoint.

IAT hooking is less common as it's ineffective in many threat scenarios. First, attackers are not obligated to use the IAT and can dynamically resolve function addresses. Second, hooks must be applied on every single PE separately and upon their loading. This presents a problem for endpoint security products which may not be installed or active at that time.

Research outline

Inline hooks are the most common hooking method in security solutions, despite being rather intrusive. You can see a community effort to map the hooks deployed by commercial products [6] and other examples such as *Cuckoo* sandbox (and its successor, *CAPE*) and *Frida* toolkit.

Techniques for bypassing and evading user-mode protection engines have existed for a very long time. Since 2020 there has been an increasing number of reports regarding malware and red teamers using them.

Curating the most comprehensive collection of these techniques led to identifying three distinct tactics attackers employ to beat those engines: hook evasion, argument forgery and engine disarming. Some techniques have variants – implementations that vary slightly but do not represent a fundamentally different approach.

The content of this paper is not arranged according to chronological order.

The detection scheme proposed in this research is derived from malware analysis and incident response investigation workflows. It lists runtime indicators to be used in real-time detection and memory artifacts for forensic analysis. The scheme is designed to be feasible and reliable, with low possibility of false positives, and robust against code obfuscation or packing.

HOOK EVASION TACTIC

The idea of hook evasion is for attackers to evade the hook. But how is this possible when the function's instructions are overwritten? As long as they can find a way to execute the original instructions and continue to the rest of the function, evasion is possible.

There are many methods to do this, which can be grouped into four classes: secondary DLL mapping, binary restoration, direct system call invocation, and code splicing.

Secondary DLL mapping

This class involves creating a separate and private instance of the DLL, on which hooks will not be placed. The malware component will use this clean copy instead of the hooked one. The class includes three techniques.

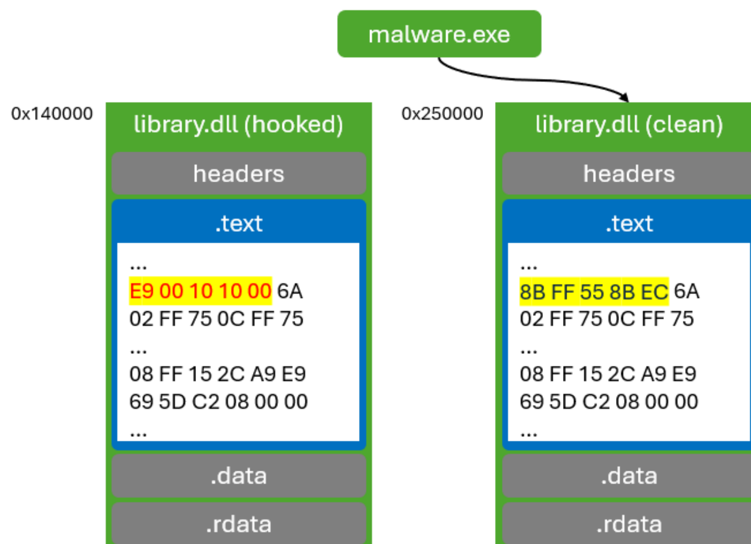

Figure 5: Illustration of secondary DLL mapping usage in a process.

1. Manually loading DLL from disk

In this technique the file is read from disk and 'reflective loading' is used so that it will be usable like other DLLs in memory.

Reflective loading is a well-known method used to run PEs without the OS's PE loader. It is used extensively by malware to execute payloads.

Avoiding the standard loading mechanism introduces detection opportunities. At runtime, a clear indicator can be found through analysis of the call stacks in the form of missing relevant DLLs. In forensics, a 'floating' PE in memory (executable memory region not mapped as image, executable pages that are private) that is a copy of a system DLL is an unambiguous artifact. Even if the PE headers are scrubbed, the code and data sections must remain.

```
0:000> kL
 # Child-SP          RetAddr               Call Site
00 0000003c`3358dbe8 00007ffe`e304b2d5     ntdll!NtCreateUserProcess+0x14
01 0000003c`3358dbf0 00007ffe`e3048316     KERNELBASE!CreateProcessInternalW+0x22d5
02 0000003c`3358f160 000001bd`e6cacef4     KERNELBASE!CreateProcessW+0x66
03 0000003c`3358f1d0 00007ff7`517d6a7d     0x000001bd`e6cacef4
04 0000003c`3358f230 00007ff7`517d2f99     malware!main+0xed
0:000> !address 0x000001bde6cacef4

Usage:                  <unknown>
Base Address:           000001bd`e6c90000
End Address:            000001bd`e6d52000
Region Size:            00000000`000c2000 ( 776.000 kB)
State:                  00001000          MEM_COMMIT
Protect:                00000040          PAGE_EXECUTE_READWRITE
Type:                   00020000          MEM_PRIVATE
Allocation Base:        000001bd`e6c90000
Allocation Protect:     00000040          PAGE_EXECUTE_READWRITE
0:000> db 0x000001bde6c90000
000001bd`e6c90000  4d 5a 90 00 03 00 00 00-04 00 00 00 ff ff 00 00  MZ..............
000001bd`e6c90010  b8 00 00 00 00 00 00 00-40 00 00 00 00 00 00 00  ........@.......
000001bd`e6c90020  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00  ................
000001bd`e6c90030  00 00 00 00 00 00 00 00-00 00 00 00 f8 00 00 00  ................
000001bd`e6c90040  0e 1f ba 0e 00 b4 09 cd-21 b8 01 4c cd 21 54 68  ........!..L.!Th
000001bd`e6c90050  69 73 20 70 72 6f 67 72-61 6d 20 63 61 6e 6e 6f  is program canno
000001bd`e6c90060  74 20 62 65 20 72 75 6e-20 69 6e 20 44 4f 53 20  t be run in DOS 
000001bd`e6c90070  6d 6f 64 65 2e 0d 0d 0a-24 00 00 00 00 00 00 00  mode....$.......
0:000> !lmi kernel32.dll
Loaded Module Info: [kernel32.dll]
         Module: KERNEL32
   Base Address: 00007ffee4090000
     Image Name: C:\WINDOWS\System32\KERNEL32.DLL
   Machine Type: 34404 (X64)
     Time Stamp: eee0fc1a (This is a reproducible build file hash, not a true timestamp)
           Size: c2000
       CheckSum: c3a18
```

Figure 6: Examining manually loading kernel32.dll from disk in WinDbg.

The drawback of this technique is quite clear – it's a significant deviation from standard OS operation, rendering it easily detectable.

The FormBook [7] malware used this technique.

2. Clone DLL

In this technique the DLL file is copied to a different path on disk and loaded using `LoadLibrary()`.

Relocations, imports and initializations are handled automatically as the OS PE loader (kernel-mode and user-mode components) is used.

Detection at runtime is possible by inspecting the call stacks, revealing unexpected DLLs that are identical to system DLLs. In forensic analysis, search for identical PEs mapped in-memory, which is highly uncommon, by comparing size, headers, sections and code (negating relocations) and module entries in the Process Environment Block's (PEB) loader lists. In addition, search the file system for duplicated system DLLs.

```
0:000> kL
 # Child-SP          RetAddr               Call Site
00 000000bd`ed2fdfb8 00007ffe`e304b2d5     ntdll!NtCreateUserProcess+0x14
01 000000bd`ed2fdfc0 00007ffe`e3048316     KERNELBASE!CreateProcessInternalW+0x22d5
02 000000bd`ed2ff530 00007ffe`9868cef4     KERNELBASE!CreateProcessW+0x66
03 000000bd`ed2ff5a0 00007ff6`270125d5     kernel32_copy!CreateProcessWStub+0x54
04 000000bd`ed2ff600 00007ff6`27012f99     malware!main+0x115
0:000> !lmi kernel32_copy.dll
Loaded Module Info: [kernel32_copy.dll]
         Module: kernel32_copy
   Base Address: 00007ffe98670000
     Image Name: C:\Windows\Temp\kernel32_copy.dll
   Machine Type: 34404 (X64)
     Time Stamp: eee0fc1a (This is a reproducible build file hash, not a true timestamp)
           Size: c2000
       CheckSum: c3a18
0:000> !lmi kernel32.dll
Loaded Module Info: [kernel32.dll]
         Module: KERNEL32
   Base Address: 00007ffee4090000
     Image Name: C:\WINDOWS\System32\KERNEL32.DLL
   Machine Type: 34404 (X64)
     Time Stamp: eee0fc1a (This is a reproducible build file hash, not a true timestamp)
           Size: c2000
       CheckSum: c3a18
```

Figure 7: Examining cloning of kernel32.dll in WinDbg.

The attacker relinquishes control over the loading procedure so the result appears similar to regular OS operation from a memory forensics perspective. The downside is that internal or lower-level dependencies can still be hooked.

Hancitor [8], SmokeLoader [9] and Winnti group [10] used this technique.

3. Section remapping

In this technique the DLL is mapped at a different base address, all while the original instance remains loaded, using `NtMapViewOfSection()`. A section handle is obtained using `CreateFile()` followed by `NtCreateSection(…, SEC_IMAGE, …)` or via `NtOpenSection()`.

This is a highly irregular operation.

Initializations such as imports resolution, DllMain and TLS callbacks are not handled because only the kernel part of OS PE loader is used. Moreover, the newly mapped section won't go through the relocation process as it is already mapped elsewhere. The reason for this is that a file on disk can have only one section representing it [11] (a one-to-one relationship).

Detection at runtime is possible by examining the call stacks for DLLs at different base addresses from those they should be at. The multiple image mappings of the same file in a single process for which corresponding module entries are missing in PEB loader lists serve as forensic artifacts.

```
0:000> kL
 # Child-SP          RetAddr               Call Site
00 0000005b`1cfaf978 00007ff7`5eaa6b1e     ntdll_16e10850000!NtWriteVirtualMemory+0x14
01 0000005b`1cfaf980 00007ff7`5eaa2f99     malware!main+0x18e
0:000> !lmi ntdll_16e10850000
Loaded Module Info: [ntdll_16e10850000]
         Module: ntdll
   Base Address: 0000016e10850000
     Image Name: ntdll.dll
   Machine Type: 34404 (X64)
     Time Stamp: 3af2a74f (This is a reproducible build file hash, not a true timestamp)
           Size: 1f8000
       CheckSum: 1fa49f
0:000> !lmi ntdll.dll
Loaded Module Info: [ntdll.dll]
         Module: ntdll
   Base Address: 00007ffee5870000
     Image Name: ntdll.dll
   Machine Type: 34404 (X64)
     Time Stamp: 3af2a74f (This is a reproducible build file hash, not a true timestamp)
           Size: 1f8000
       CheckSum: 1fa49f
```

Figure 8: Examining section remapping of ntdll.dll in WinDbg.

The pitfall of this technique is that it can't be used for complex code because relocations and initializations are not handled. Attempting to manually apply relocations will introduce private pages of code, resembling reflective loading and making it conspicuous in memory forensics.

HawkEye's loader [12], the TxHollower [13, 14] and HijackLoader [15, 16] malware families used this technique.

This class of methods is also effective for evading hardware breakpoints.

We see that all these techniques can be accurately detected both at runtime and during forensic analysis.

Binary restoration \ unhooking

This class involves unhooking functions by manipulating their contents in memory. In essence, the inverse operation to installing hooks. This class includes four techniques, with a total of nine variants.

1. Temporary copy

In this technique a clean, hook-free instance of the DLL is created in memory, the code is copied back from it, and it is removed afterwards. This technique is called 'bindiff and unhook' by some.

The techniques for secondary mapping (the previous class) already provide the original code, hence this technique has three variants:

1. Manually load DLL from disk (reflective loading)

   To create an accurate copy, the relocations are applied according to the base address of the target DLL.

   This is also referred to as 'Unhook Flashbang' [17] or 'Reflective DLL Refresh' [18]. LockBit ransomware affiliates [19] and HijackLoader [15, 16] used this variant.

2. Clone DLL

   After `LoadLibrary` returns, relocations need to be applied according to the base address of the target DLL.

   The Winnti group [10], which is also referred to as APT41 [20], along with another Chinese state-sponsored threat actor [21] used this variant.

3. Section remapping [22]

   There is no need to handle relocations since they are already there due to OS operation.

   APT29 [23], HijackLoader [15, 16] and a Chinese-linked APT group [24] used this variant.

Next, `VirtualProtect()` is used to make the code writeable and the code is overwritten back to restore it.

Some implementations also recursively verify the IAT and the imported functions themselves.

When using the clone DLL variant there will be multiple mappings of identical PEs (different files on disk with matching content). Using the section remapping variant will cause multiple mappings of the same PE (the same file on disk). These runtime indicators, in conjunction with call stacks without any discrepancies, can be used as runtime indicators. Repeated file access to system DLLs, while unusual, is not a conclusive indicator on its own.

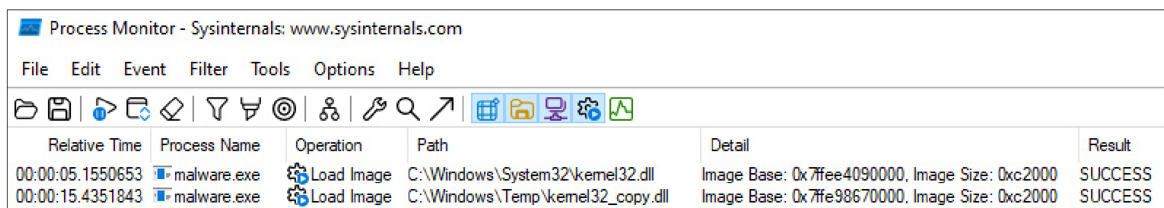

Figure 9: Multiple load image events in Process Monitor when cloning kernel32.dll.

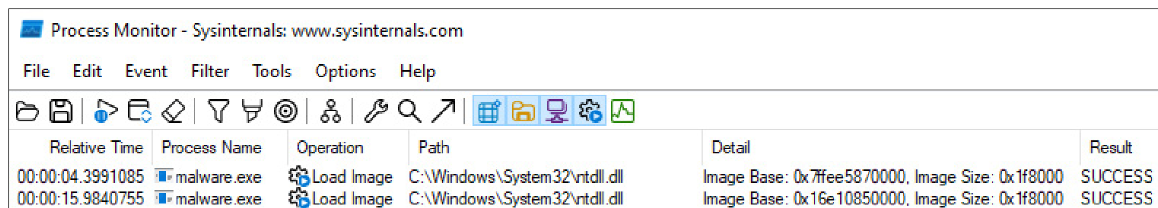

Figure 10: Recurring load image events in Process Monitor when remapping ntdll.dll.

2. Peer ripping

In this technique the content of the DLL is copied from the memory of a different process instead of accessing the file system directly.

A quick guide on the process lifecycle: when a new process is created, its memory space is initialized with the main executable and ntdll.dll mapped into it. Processes are created in suspended state. At this point, the protection engine hasn't loaded and applied its hooks yet. The process is resumed and starts from ntdll.dll to load the DLLs in the import table. Once that is done and all imports have been handled, the queued user APCs are executed. Afterwards the main thread transitions to the executable's entrypoint.

This technique has three variants:

1. Suspended child process

   A child process is created that is suspended with `CreateProcess(…, CREATE_SUSPENDED, …)`. This is applicable only for ntdll.dll.

   The creator of this method named it 'Perun's Fart' [25].

2. Debugged child process

   A child process is created with `CreateProcess(…, DEBUG_PROCESS, …)` and debugged using `WaitForDebuggerEvent()`. Either a hardware breakpoint is set with `SetThreadContext()` at the loader functions (named 'Blindside' [26] by its creator) or the `LOAD_DLL_DEBUG_EVENT` is used.

   This pauses the process execution right as the DLL is mapped to memory and before the protection engine can apply its hooks.

3. Existing process [27]

   `OpenProcess()` is used to gain access to another process. Depending on the granted access permissions `RtlCreateProcessReflection()` or `PssCaptureSnapshot()` might need to be called to get read permissions.

This is possible since not all processes are actively monitored by security solutions – such as excluded system processes, for example.

Then, the content is read with `NtReadVirtualMemory()`, and `VirtualProtect()` is used to make the code writeable before overwriting it back.

The procedure across all variants is only momentary so the chance to detect it in forensics is slim to none.

Creation of a debugged child process by a parent that is not a known debugger can serve as a runtime indicator. A suspended process creation is not unique so it's unreliable as a runtime indicator. While opening process handles randomly is uncommon, on its own it is not enough to accurately deduce that hook evasion has been attempted. Reading executable memory from another process is unusual, but there is no standard mechanism to intercept (and prevent) it in real time.

3. Section refresh [28]

In this technique, instead of overwriting the code, the DLL is mapped again in place. This restores the original code and avoids the hooks being reapplied as the mapping is done manually, without the protection engine's awareness or control. This is conceptually similar to 'section remapping'.

This technique consists of three steps:

1. Capture: the writeable, non-shared PE sections are copied and the memory protections of each section are saved.
2. Refresh: an image section handle is obtained with `NtCreateSection(…, SEC_IMAGE, …)`, the current mapping is removed from the address space with `NtUnmapViewOfSection(…, module_base, …)`, and the image is mapped back to memory using `NtMapViewOfSection(…, module_base)`.
3. Restore: the PE loader's operation is completed by filling the imports (the IAT), forwarder exports and CFG validator function pointer, and copying back the state saved in step one.

Repeated mappings of the same PE on the same base address is a solid runtime indicator, as that is highly uncommon.

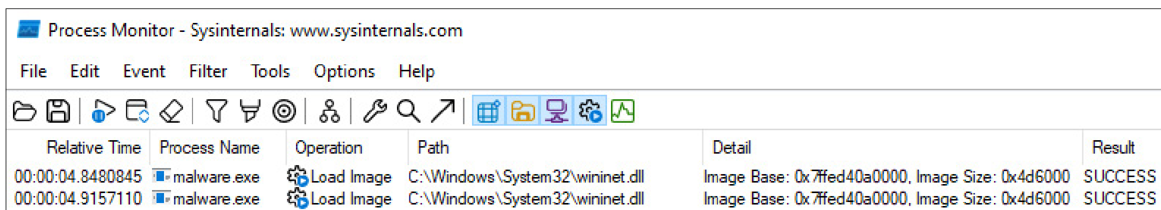

Figure 11: Repeating load image events in Process Monitor when refreshing wininet.dll.

This method can't be used on ntdll.dll as it is dependent on its APIs. In multi-threaded processes there is a risk of instability as DLLs are a global resource in a process. If other threads interact with the same DLL during the procedure it may lead to access violations or inconsistent state-related issues.

4. Short-circuiting

As the original instructions must be kept in memory for the hooked function to work properly, in this technique, the hook is redirected to them. The protection engine will be averted even as the instructions of the target function remain modified.

To locate the original instructions, the process address space is searched, using `NtQueryVirtualMemory()`, for regions that are committed, private and executable (`MEM_COMMIT` | `MEM_PRIVATE` | `PAGE_EXECUTE`). Each such region is scanned for the control flow instruction back to the target function.

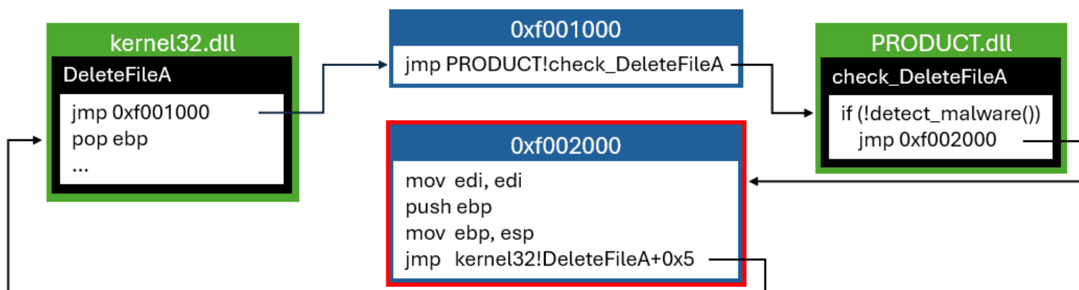

Figure 12: Illustration of the function's original instructions with copy region highlighted.

Once the original instructions are found, there are two options (variants):

1. Overwrite the hook itself [29]

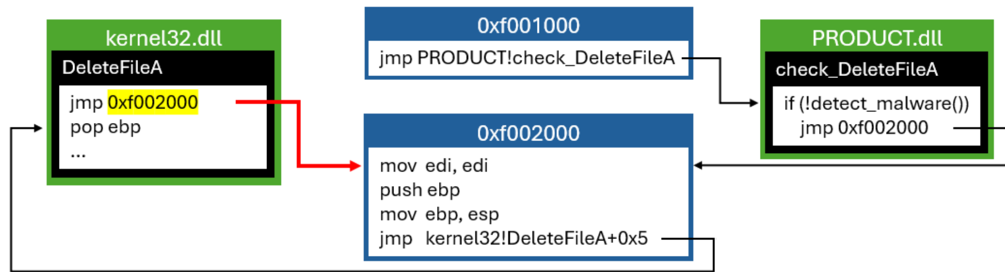

Figure 13: Illustration of short-circuiting the hook.

2. Overwrite the trampoline [30]

   Trampolines aren't always present with every hook. They are an internal layer which is often overlooked and sometimes kept in writeable-executable (WX) memory.

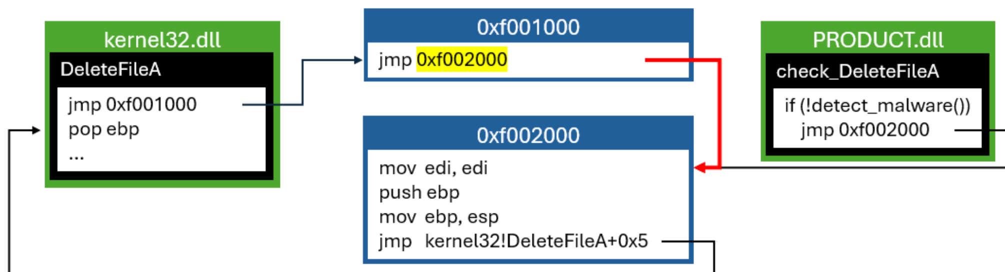

Figure 14: Illustration of short-circuiting the trampoline.

`VirtualProtect()` is used to make the code writeable.

If scanning for the original instructions region fails, disassembling and statically tracking the code flow from the hook can also lead to them, though it is considerably more intricate.

There are no indicators for detection at runtime. The altered hooks are the artifact to identify in forensics investigation.

The runtime indicators are more subtle as there are no changes in the call stack.

For forensic artifacts, these techniques leave a clear trace since, except for short-circuiting, the hooks are removed, which is straightforward to detect. A new *Volatility* plugin [31] attempts to do just that. As mentioned, it is also easy to identify the modified hooks when short-circuiting is used.

None of the techniques so far can escape detection in forensics and the majority of them are susceptible to detection at runtime as well. Aside from that, the problem with all the methods so far is that they are dependent on system calls, which themselves can be hooked.

Direct system call invocation

This class uses the `syscall` \ `sysenter` \ `int 2e` instructions and implements the OS native interface instead of calling the APIs in ntdll.dll. These instructions can also be found in user32.dll or win32u.dll (introduced with *Window 10*).

The calling convention for system service calls on *Windows* requires setting the index of the requested service in the eax register. Parameters are passed as in the stdcall calling convention on 32-bit processes and fastcall on 64-bit processes.

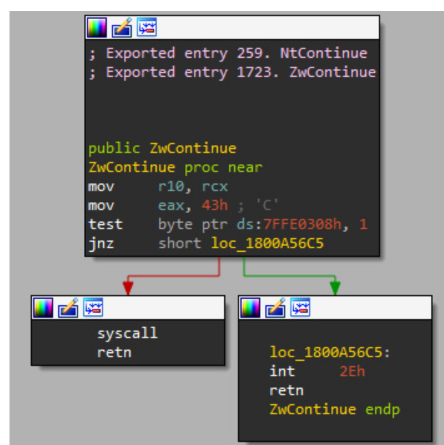

Figure 15: Disassembly graph of NtContinue system service function in ntdll.dll.

`int 2e` used to be the legacy system service call transition mechanism. In 2016, it was reintroduced with the release of Virtualization-Based Security (VBS) on *Windows 10*, and made the default mechanism when VBS is enabled on the system. VBS is the foundation for capabilities like Hypervisor-protected Code Integrity (HVCI).

There are five techniques with eight variants in total to obtain the system service indexes.

1. ntdll.dll parsing

In this technique the index of the desired system service is resolved by static analysis of ntdll.dll. This technique has three variants:

1. Parse instructions (disassemble\pattern matching)

   The desired system service function is scanned for the opcode of the `mov eax` instruction and the operand value is extracted from it. That can be done from the file on disk or in-memory (similar to 'peer ripping' but not agnostic).

   A researcher published a comprehensive blog post on this variant back in 2014 [32]. In-memory parsing is also dubbed 'Hell's Gate' [33], but this publication assumes functions are not hooked.

   The Floki Bot dropper [34], GlobeImposter ransomware [35], LockPoS [36], TrickBot [37], DarkGate [38] (based on open-source Delphi code from 2012 [39]), Gootkit [40], Parallax RAT [41], BabLock [42]\Rorschach [43] ransomware, Lumma stealer [44] and Cobalt Strike stagers in Hive ransomware attacks [45] all used this variant.

2. Count functions

   The system service functions are sorted in ascending order inside ntdll.dll. The exported Nt functions are traversed to create a sorted list and the number of items before the desired system service are counted instead of decoding opcodes.

   The FreshyCalls [46] and SysWhispers2 [47] projects implement this.

3. Proximity check

   Because the system service functions are in the binary in ascending order it's possible to go straight to the hooked function and parse the index of its neighbouring system service function (preceding, successor or both) and increment or subtract it. If the neighbour is hooked as well, the action is simply repeated until a non-hook function is found.

   This variant is also called 'Neighbor Peek'.

   The Halo's Gate [48] and Tartarus' Gate [49] projects implement this variant.

```
0:000> u ntdll!NtSetEvent
ntdll!NtSetEvent:
00007ffe`107ad6a0 4c8bd1          mov     r10,rcx
00007ffe`107ad6a3 b80e000000      mov     eax,0Eh
00007ffe`107ad6a8 f604250803fe7f01 test    byte ptr [SharedUserData+0x308],1
00007ffe`107ad6b0 7503            jne     ntdll!NtSetEvent+0x15
00007ffe`107ad6b2 0f05            syscall
00007ffe`107ad6b4 c3              ret
00007ffe`107ad6b5 cd2e            int     2Eh
00007ffe`107ad6b7 c3              ret
ntdll!NtClose:
00007ffe`107ad6c0 4c8bd1          mov     r10,rcx
00007ffe`107ad6c3 b80f000000      mov     eax,0Fh
00007ffe`107ad6c8 f604250803fe7f01 test    byte ptr [SharedUserData+0x308],1
00007ffe`107ad6d0 7503            jne     ntdll!NtClose+0x15
00007ffe`107ad6d2 0f05            syscall
00007ffe`107ad6d4 c3              ret
00007ffe`107ad6d5 cd2e            int     2Eh
00007ffe`107ad6d7 c3              ret
ntdll!NtQueryObject:
00007ffe`107ad6e0 4c8bd1          mov     r10,rcx
00007ffe`107ad6e3 b810000000      mov     eax,10h
00007ffe`107ad6e8 f604250803fe7f01 test    byte ptr [SharedUserData+0x308],1
00007ffe`107ad6f0 7503            jne     ntdll!NtQueryObject+0x15
00007ffe`107ad6f2 0f05            syscall
00007ffe`107ad6f4 c3              ret
00007ffe`107ad6f5 cd2e            int     2Eh
00007ffe`107ad6f7 c3              ret
ntdll!NtQueryInformationFile:
00007ffe`107ad700 4c8bd1          mov     r10,rcx
00007ffe`107ad703 b811000000      mov     eax,11h
00007ffe`107ad708 f604250803fe7f01 test    byte ptr [SharedUserData+0x308],1
00007ffe`107ad710 7503            jne     ntdll!NtQueryInformationFile+0x15
00007ffe`107ad712 0f05            syscall
00007ffe`107ad714 c3              ret
00007ffe`107ad715 cd2e            int     2Eh
00007ffe`107ad717 c3              ret
```

Figure 16: Disassembly of consecutive system service functions in ntdll.dll in WinDbg.

2. Heaven's Gate

Heaven's Gate is a well-known method published in 2009 [50, 51] for a 32-bit process to switch to 64-bit execution context on a 64-bit *Windows* system without using the system DLLs that implement the WOW64 emulation layer [52]. It enables 32-bit applications to use OS APIs on 64-bit processes.

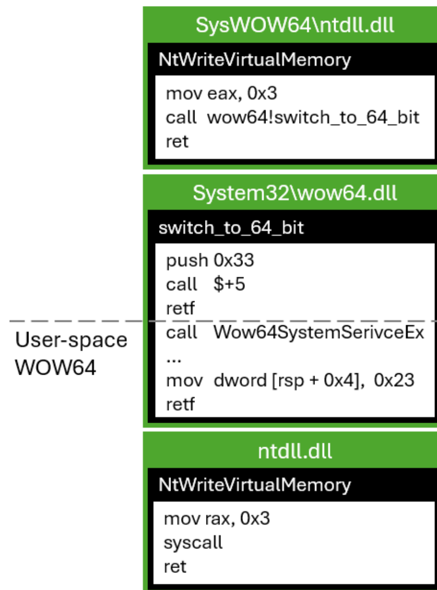

Figure17: Illustration of a 32-bit process system service call on 64-bit Windows.

Using Heaven's Gate allows user-mode hooks to be evaded in couple of ways (variants):

1. Skip WOW64 layer

   By directly calling functions of the 64-bit ntdll.dll the hooks placed in the 32-bit ntdll.dll are skipped over. This is due to protection engines reusing hooks in the 32-bit context despite the native interface actually being implemented in the 64-bit context.

Figure 18: Illustration of Heaven's Gate skip WOW64 layer variant.

Vawtrack [53], Ursnif (Gozi) [54], Nymaim [55], crypto miner [56], TrickBot [37], Emotet [57] and Lumma stealer [44] used this variant.

Interestingly, Lumma stealer implemented the variant only partially – just the 32-bit ntdll.dll is skipped over, with the call still going through the rest of the WOW64 DLLs, meaning it risks monitoring by hooks placed in these DLLs.

2. Completely native

   Directly perform the transition to the kernel from the 64-bit context after using any technique from this class to get the system service index.

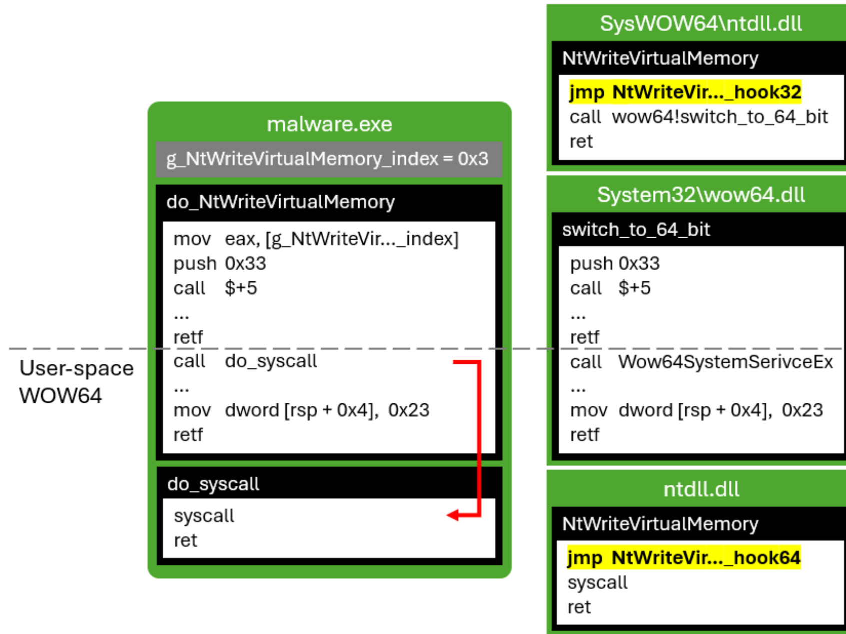

Figure 19: Illustration of Heaven's Gate completely native variant.

Note: debuggers and analysis tools are usually blind to the occurrences of these architectural transitions without an explicit update.

Floki Bot dropper [34], GlobeImposter ransomware [35], Gootkit [40], Pony stealer [58], an unnamed loader for RATs and stealers [59], TxHollower [13], GuLoader [60] with the BYOI method (described later in this class) and HijackLoader [15, 16] all used this variant.

3. ntoskrnl.exe parsing [61]

In this variant ntoskrnl.exe is parsed, in user-space, without special privileges, instead of ntdll.dll to resolve indexes of system services.

In kernel mode, the Nt function is the actual implementation of the system service and the Zw function version has the index. These functions are exported by ntoskrnl.exe. `GetProcAddress()` is called to find them and the functions are scanned for the `mov eax` instruction, as before.

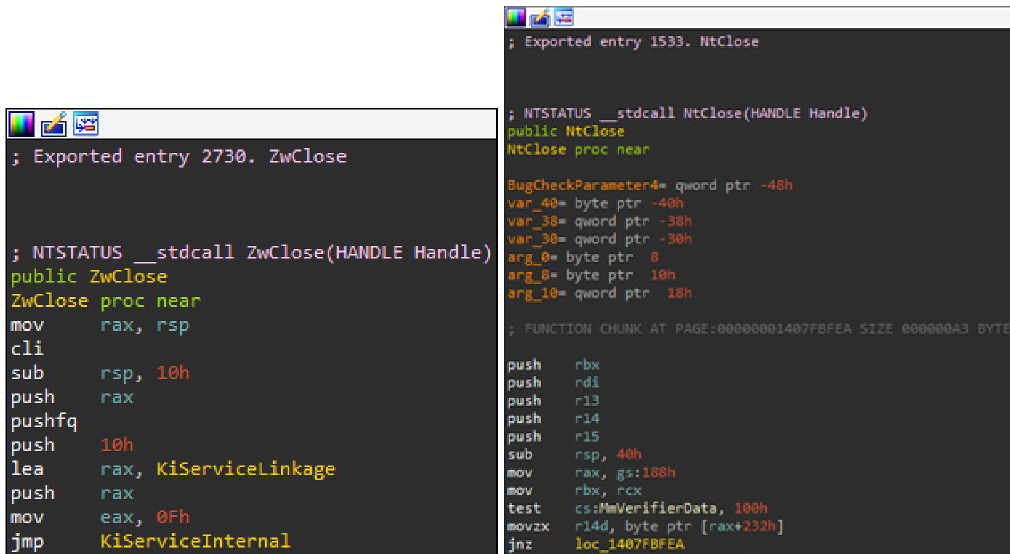

Figure 20: Disassembly of exported Zw and Nt functions in ntoskrnl.exe.

Some system service functions only have the Nt version exported. For them, the system service index can be obtained from the System Service Descriptor Table (SSDT) array (`KiServiceTable` symbol). This is an undocumented and internal data structure. As it's not exported, it first needs to be located.

```
.rdata:00000001400C7A60 KiServiceTable  dd rva NtAccessCheck
.rdata:00000001400C7A64                 dd rva NtWorkerFactoryWorkerReady
.rdata:00000001400C7A68                 dd rva NtAcceptConnectPort
.rdata:00000001400C7A6C                 dd rva NtMapUserPhysicalPagesScatter
.rdata:00000001400C7A70                 dd rva NtWaitForSingleObject
.rdata:00000001400C7A74                 dd rva NtCallbackReturn
.rdata:00000001400C7A78                 dd rva NtReadFile
.rdata:00000001400C7A7C                 dd rva NtDeviceIoControlFile
.rdata:00000001400C7A80                 dd rva NtWriteFile
.rdata:00000001400C7A84                 dd rva NtRemoveIoCompletion
.rdata:00000001400C7A88                 dd rva NtReleaseSemaphore
.rdata:00000001400C7A8C                 dd rva NtReplyWaitReceivePort
.rdata:00000001400C7A90                 dd rva NtReplyPort
.rdata:00000001400C7A94                 dd rva NtSetInformationThread
.rdata:00000001400C7A98                 dd rva NtSetEvent
.rdata:00000001400C7A9C                 dd rva NtClose
.rdata:00000001400C7AA0                 dd rva NtQueryObject
.rdata:00000001400C7AA4                 dd rva NtQueryInformationFile
.rdata:00000001400C7AA8                 dd rva NtOpenKey
```

Figure 21: The NtClose function in KiServiceTable in ntoskrnl.exe.

First, a number of Nt and Zw function pairs need to be identified, after which the system service indexes are parsed and the offsets of the Nt functions are computed from the image base. The data sections are scanned for an array with entries of matching offset in each index. This array is the SSDT.

```
#define MAX_NUMBER_OF_SYSCALLS 512

PDWORD find_service_table_base(HMODULE ntos)
{
    PDWORD service_table_base = NULL;

    DWORD NtReadFile_offset = GetProcAddress(ntos, "NtReadFile") - ntos;
    DWORD NtWriteFile_offset = GetProcAddress(ntos, "NtWriteFile") - ntos;
    DWORD NtClose_offset = GetProcAddress(ntos, "NtClose") - ntos;

    DWORD ZwReadFile_index = parse_ntos_zw_export_for_syscall_index(GetProcAddress(ntos, 
"ZwReadFile"));
    DWORD ZwWriteFile_index = parse_ntos_zw_export_for_syscall_index(GetProcAddress(ntos, 
"ZwWriteFile"));
    DWORD ZwClose_index = parse_ntos_zw_export_for_syscall_index(GetProcAddress(ntos, "ZwClose"));

    DWORD highest_index = max(max(ZwReadFile_index, ZwWriteFile_index), ZwClose_index);

    PDWORD service_table = malloc((highest_index + 1) * sizeof(DWORD));
    memset(service_table, 0, (highest_index + 1) * sizeof(DWORD));

    service_table[ZwReadFile_index] = NtReadFile_offset;
    service_table[ZwWriteFile_index] = NtWriteFile_offset;
    service_table[ZwClose_index] = NtClose_offset;

    PIMAGE_NT_HEADERS nt_headers = RtlImageNtHeader(ntos);
    PVOID service_table_search_limit = 
        RVA_TO_VA(ntos, (nt_headers->OptionalHeader.SizeOfImage - (sizeof(DWORD) * MAX_NUMBER_OF_
SYSCALLS)));

    for (ULONG_PTR current_ntos_address = (ULONG_PTR)RVA_TO_VA(ntos, nt_headers->OptionalHeader.
SizeOfHeaders);
         ((current_ntos_address < (ULONG_PTR)service_table_search_limit) && (service_table_base == 
NULL));
         current_ntos_address += sizeof(DWORD))
    {
         BOOL matched_all_entires = FALSE;
```

```
        PDWORD current_service_table_base = (PDWORD)current_ntos_address;

#if (CURRENT_ARCHITECTURE == IMAGE_FILE_MACHINE_I386)
        matched_all_entires =
            service_table[ZwReadFile_index] == (DWORD)RVA_TO_VA(ntos, current_service_table_
base[ZwReadFile_index]) &&
            service_table[ZwWriteFile_index] == (DWORD)RVA_TO_VA(ntos, current_service_table_
base[ZwWriteFile_index]) &&
            service_table[ZwClose_index] == (DWORD)RVA_TO_VA(ntos, current_service_table_
base[ZwClose_index]);
#elif (CURRENT_ARCHITECTURE == IMAGE_FILE_MACHINE_AMD64)
        matched_all_entires =
            service_table[ZwReadFile_index] == current_service_table_base[ZwReadFile_index] &&
            service_table[ZwWriteFile_index] == current_service_table_base[ZwWriteFile_index] &&
            service_table[ZwClose_index] == current_service_table_base[ZwClose_index];
#endif

        if (matched_all_entires)
        {
            service_table_base = current_service_table_base;
        }
    }

    return (service_table_base);
}
```

Figure 22: C-pseudocode for finding KiServiceTable in ntoskrnl.exe.

Now, the offset of the desired exported Nt function can be calculated from the image base and the table traversed to find reference to it. The index of the entry is the system service index.

```
DWORD find_system_service_index_for_nt_function(HMODULE ntos, char * nt_function_name)
{
    DOWRD index = MAX_DWORD;

    PVOID nt_function_address = GetProcAddress(ntos, nt_function_name);

    for (DWORD current_index = 0;
         ((current_index < MAX_NUMBER_OF_SYSCALLS && (index == MAX_DWORD));
         ++current_index)
    {
        PVOID current_system_service_function;

#if(CURRENT_ARCHITECTURE == IMAGE_FILE_MACHINE_I386)
        system_service_function = (PVOID)ki_service_table[current_index];
#elif(CURRENT_ARCHITECTURE == IMAGE_FILE_MACHINE_AMD64)
        system_service_function = (PVOID)RVA_TO_VA(ntos, ki_service_table[current_index]);
#endif

        if (system_service_function == nt_function_address)
        {
            index = current_index;
        }
    }

    return (index);
}
```

Figure 23: C-pseudocode for getting a system service index from KiServiceTable.

The number of exported system services in ntoskrnl.exe is approximately half the number of those in ntdll.dll, though still handy: ZwCreate*, ZwQuery\Set*, ZwReadFile, ZwWriteFile, ZwMap\UnmapViewOfSection, ZwAllocate\Protect\FreeVirtualMemory, ZwLoad\UnloadDriver, ZwYieldExecution (Sleep), Registry API, Transactions API.

On some *Windows 10* releases, certain *Microsoft* processes, like explorer.exe, statically imported ntoskrnl.exe.

4. Bring Your Own Index (BYOI) [62]

In this technique, rather than parsing PEs, the payload brings with it the indexes of all required system services.

The indexes are not guaranteed to be the same across different *Windows* builds. Thus, each system service index is also required per each specific OS version the attacker targets.

This creates a maintenance burden to support multiple OS versions and constantly verify each new major OS update. A security researcher developed such a project and made it available for the community [63], effectively taking care of this heavy lifting. It's necessary for the malware to have an update mechanism to avoid becoming non-functional subsequent to OS updates.

The SysWhispers [64] project implements this technique.

The Chimera APT group [65] and GuLoader [60] used this technique.

5. Dynamic resolution

In this technique, instead of parsing or hard-coding the index values, the system service interface function is allowed to run and then stopped before transitioning to kernel-mode where the calling convention has already been satisfied.

The technique involves the following steps:

a. An exception handler is set up using `SetUnhandledExceptionFilter()` [66] or `RtlAddVectoredExceptionHandler()`.

b. The user-to-kernel transition instruction (`syscall` \ `int 2e`) is located in the target system service function and a breakpoint placed on it.

c. The system service function is called.

d. The eax value is retrieved from the `CONTEXT` structure in the exception handler, and the breakpoint can be removed.

Deploying a hardware breakpoint avoids memory manipulation. Single-stepping is a viable alternative as well. In this case, the second step would change to start it instead of setting a breakpoint and the exception handler would end it in the last step.

Direct system call invocation results in missing system DLLs from call stacks. With the exception of Heaven's Gate, ntdll.dll won't be present in the call stacks. Although this is a straightforward indicator for detection at runtime, it does not allow for the distinguishing of which specific technique was used.

```
0:000> kL
 # Child-SP          RetAddr               Call Site
00 00000004`86fdf6d8 00007ff6`b9572c39     malware!main+0xa6
0:000> u malware!main+0x9c
malware!main+0x9c
00007ff6`b9572357 4c8bd1          mov     r10,rcx
00007ff6`b957235a b83a000000      mov     eax,3Ah
00007ff6`b957235f 0f05            syscall
```

Figure 24: Examining direct system call invocation usage in WinDbg.

Call stacks will be a bit different with Heaven's Gate. They only miss the WOW64 system DLLs with the skip WOW64 layer variant.

```
0: kd> !wow64exts.k
Walking Native Stack...
 # Child-SP          RetAddr               Call Site
00 fffff885`32f9ca88 fffff805`7e211f05     nt!NtWriteVirtualMemory
01 fffff885`32f9ca90 00007ffd`0e5adc34     nt!KiSystemServiceCopyEnd+0x25
02 00000000`005ce038 00007ffd`0e2cf0c1     ntdll!NtWriteVirtualMemory+0x14
03 00000000`005ce040 00007ffd`0e2c90da     wow64!whNtWriteVirtualMemory+0x51
04 00000000`005ce090 00000000`775c17c3     wow64!Wow64SystemServiceEx+0x15a
05 00000000`005ce950 00000000`775c11b9     wow64cpu!ServiceNoTurbo+0xb
06 00000000`005cea00 00007ffd`0e2c3989     wow64cpu!BTCpuSimulate+0x9
07 00000000`005cea40 00007ffd`0e2c337d     wow64!RunCpuSimulation+0xd
08 00000000`005cea70 00007ffd`0e5e3631     wow64!Wow64LdrpInitialize+0x12d
09 00000000`005ced20 00007ffd`0e585deb     ntdll!LdrpInitializeProcess+0x18e1
0a 00000000`005cf140 00007ffd`0e585c73     ntdll!LdrpInitialize+0x15f
0b 00000000`005cf1e0 00007ffd`0e585c1e     ntdll!LdrpInitialize+0x3b
0c 00000000`005cf210 00000000`00000000     ntdll!LdrInitializeThunk+0xe
Walking Guest (WoW) Stack...
 # ChildEBP          RetAddr
00 008ff738 001e1bdc      ntdll_775d0000!NtWriteVirtualMemory+0xc
01 008ff86c 001e4e23      app!main+0x10c
```

Figure 25: Examining usage of Nt function in WOW64 ntdll.dll in WinDbg.

```
0: kd> !wow64exts.k
Walking Native Stack...
 # Child-SP          RetAddr               Call Site
00 fffff885`30e9ea88 fffff805`7e211f05     nt!NtWriteVirtualMemory
01 fffff885`30e9ea90 00007ffd`0e5adc34     nt!KiSystemServiceCopyEnd+0x25
02 00000000`0133f6a8 00000000`00963f14     ntdll!NtWriteVirtualMemory+0x14
03 00000000`0133f6b0 cccccccc`cccccccc     malware!X64Call+0x214
Walking Guest (WoW) Stack...
 # ChildEBP          RetAddr
00 0133efd8 77601d11      ntdll_775d0000!NtAllocateVirtualMemory+0xc
```

Figure 26: Examining usage of Heaven's Gate skip WOW64 layer variant in WinDbg.

Call stacks will miss WOW64 system DLLs and ntdll.dll entirely with the Heaven's Gate completely native variant.

```
0: kd> !wow64exts.k
Walking Native Stack...
 # Child-SP          RetAddr               Call Site
00 fffff885`30e9ea88 fffff805`7e211f05     nt!NtWriteVirtualMemory
01 fffff885`30e9ea90 00000000`00964327     nt!KiSystemServiceCopyEnd+0x25
02 00000000`0133f7c0 00961aa0`00000000     malware!X64SyscallCall+0x227
03 00000000`0133f7c8 0133f8a0`0133f99c     0x00961aa0`00000000
Walking Guest (WoW) Stack...
 # ChildEBP          RetAddr
00 0133f9e4 00961a21      ntdll_775d0000!NtClearEvent+0xc
```

Figure 27: Examining usage of Heaven's Gate completely native variant in WinDbg.

On systems with VBS enabled, an additional runtime indicator presents itself when drilling down the call stack – the `syscall` instruction is used instead of the `int 2E` instruction as most implementations don't accurately handle that switch.

Searching for `syscall` \ `sysenter` \ `int 2e` instructions outside of the OS DLLs as a forensic artifact might not yield high detection rates. It requires disassembling of large amounts of code, which can be defeated by packing or anti-static analysis tricks like code obfuscations. The new *Volatility* plugin [31], mentioned earlier, does attempt that, however.

The downside of these techniques is that they can't be used generically as they are tailored for a platform-dependent interface.

Apart from the Heaven's Gate completely native variant, all other techniques in this class can be utilized for binary restoration as well. GuLoader [60] used the counting variant of ntdll.dll parsing and VBCrypter [67] used the proximity check variant to unhook ntdll.dll.

This class of methods is also effective for evading hardware breakpoints.

The problem with all techniques so far is that they are (still) noisy and detectable in some capacity.

Code splicing \ byte stealing

In this class the function prolog is rebuilt elsewhere in memory and run instead of the hook in the target function.

Figure 28: Illustration of code splicing kernel32.dll and ntdll.dll functions.

Two techniques exist:

1. From disk

   In this technique 'Manually Loading DLL From Disk' is leveraged to extract the target function instructions instead of using it as a secondary instance or for unhooking. Section remapping, clone DLL and peer repping may also be used but they are noisier.

The code can also be placed in the malware's binary region.

This technique was in common use by legacy packers and code protectors. CodeFork's Gamarue [68] malware used this technique.

2. Stub reuse

   If the target function is already hooked, the original instructions were kept in memory, as established by the short-circuiting technique. The advantage of this technique over short-circuiting is that no changes to memory are required.

   The FireWalker [69] project implements this concept, albeit slightly differently.

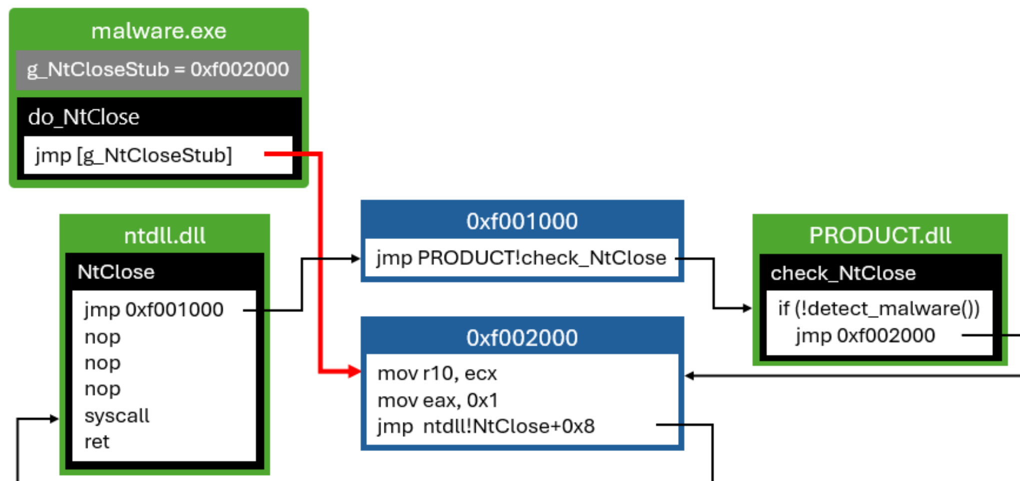

Figure 29: Illustration of code splicing sub reuse variant of NtClose function.

Code splicing applied on system service functions can be regarded as 'indirect system call invocation'. Since it's a specifically tailored solution this research doesn't consider it as a technique on its own. Pikabot [70] and SquidLoader [71] used variants of ntdll.dll parsing to perform that. The former employed counting. The latter parsed instructions in memory (Hell's Gate) and defaulted to proximity check (Neighbor Peek) if a target function was hooked.

Neither technique has traces in runtime or forensic artifacts. On runtime, the call stacks won't have irregularities. A code page that is private and executable is not meaningful by itself in forensics investigation. Examining the contents of such pages for the original function stubs won't cut it either, since in forensics there's really no way to tell who created it – the malware or the protection engine. Multiple stubs of a function can exist in memory due to multiple security tools running on the device that are incompatible. That is not an uncommon scenario. Regardless, this artifact doesn't cover the sub reuse technique.

This class of methods is also effective for evading hardware breakpoints.

The flaw of these techniques is that internal or lower-level dependencies and functions may still be hooked.

ARGUMENTS FORGERY TACTIC

Attackers won't attempt to evade the hooks directly. Instead, they feed the protection engine with bogus arguments and swap them after inspection but before they are used by the target function. This is a known class of software bugs called 'Time-Of-Check to Time-Of-Use' (TOCTOU) [72].

Arguments forgery is an emerging approach introduced a couple of years ago.

The technique leverages control flow modification constructs, i.e. interrupts, to create an opportunity to change the arguments without modifying the hook or protection engine. There are couple of options to halt the execution flow this way:

1. Hardware breakpoint [73].
2. An argument that will cause an exception in the protection engine [74]. This is vendor-dependent so it's not a reliable option.

It is implemented as follows:

a. An exception handler is set up with `SetUnhandledExceptionFilter()` or `RtlAddVectoredExceptionHandler()`.

b. The address of the instruction the protection engine returns to in the target function is found and a breakpoint is placed on it.

c. The target function is called with fake values. The hook and the protection engine's logic are passed through (TOC). In due course, the protection engine calls back the monitored function.

d. The parameters are changed to the real values according to the target function's calling convention (registers in the `CONTEXT` structure and on the thread's stack) and the breakpoint is removed by the exception handler.

e. Execution continues (TOU).

```
0:000> u malware!main+0xe5
malware!main+0xe5:
00007ff6`d7a91c15 ff151df60000    call    qword ptr [malware!_imp_InternetOpenUrlA]
0:000> da rdx
00007ff6`b056ad10  "https://benign-domain.com"
```

```
0:000> u wininet!InternetOpenUrlA+0x7 L2
WININET!InternetOpenUrlA+0x7:
00007ffe`d41d7f37 48896810        mov     qword ptr [rax+10h],rbp
00007ffe`d41d7f3b 48897018        mov     qword ptr [rax+18h],rsi
0:000> da rdx
00007ff6`d7a9ace0  "https://malicious-domain.com"
```

Figure 30: Arguments before call to hooked function (above) and after modification (below).

Single-stepping can be used in this case as well. Instead of setting a breakpoint, the second step would change to start it and the exception handler will end it in the step before last.

This technique basically advances the dynamic resolution method one step further. The concept of fake arguments itself isn't novel. It has been used against sandboxes and monitoring tools to flood them with useless random data.

Without an attached debugger, a hardware breakpoint in a hooked function after the hook or the trap flag set in the thread's flags register right before the engine calls back to the target function are viable indicators to detect at runtime. With the target function being a system service, they won't allow a distinction between this technique and dynamic resolution. Therefore, they were not suggested for detection of the latter. Forensic artifacts are not available since this operation is short-lived.

This technique has a major shortcoming – the protection engine is still invoked, which means it has a chance to not only detect this technique but potentially block it.

ENGINE DISARMING TACTIC

This idea is self-explanatory – the protection engine is targeted in the process address space instead of removing a specific hook or evading the detection logic. There are several methods to achieve this:

1. Unload the engine DLL

   Use of `FreeLibrary()` [75] or triggering the engine's unload function (vendor-specific). Once the engine is unloaded it either uninstalls the hooks or disables its monitoring code.

2. Unmap all DLLs [76]

   This is like 'section refresh', just without the 'refresh' step. This method is quite problematic as it puts the process in a broken state. A process without ntdll.dll may have difficulties continuing to run properly.

3. Allow only loading of *Microsoft*-signed binaries [77, 78]

   A new process is spawned with Code Integrity Guard (CIG) and Arbitrary Code Guard (ACG) mitigations enabled, leveraging the OS capabilities to prevent the protection engine's DLL from loading.

4. Preloading in new child process

   This is a relatively recent approach to block the protection engine from initializing in a new process by controlling its lifecycle (explained in the 'peer ripping' technique):

   a. Debugged process [79] – break on image load and modify the DLL entrypoint routine to do nothing, rendering the engine inoperable.

   b. Inject the process – run inside the process before the protection engine. AppVerifier [80] and Shim Engine [81] callbacks can be used to execute code at the earliest possible stage. The injected code hooks ntdll.dll's `KiUserApcDispatcher` and `LdrLoadDll` functions to suppress the engine's loading.

Additional vendor-specific implementation issues can also be leveraged by attackers (e.g. toggling internal state to disable hooks [82]). However, since they are not generic and broadly applicable, they are not covered here.

SUMMARY

Attackers continually study how security tools and products work and adjust themselves accordingly. This paper documented 26 unique methods to beat user-mode-based protection engines, clearly demonstrating this is the most prolific post-exploitation technique to date, surpassing even code injection methods.

These techniques are trivial to implement and simple to use. You can find source code for most of them publicly available online. There are overlaps between these techniques as they are built on similar approaches and share certain primitives.

Many of these techniques are used in the wild, with the most prevalent being direct system call invocation after parsing ntdll.dll instructions and Heaven's Gate completely native variant. Usage of Heaven's Gate will likely decrease in the future with native 32-bit systems becoming less popular. HijackLoader employs the largest number of techniques by a single malware.

The MITRE ATT&CK framework 'Impair Defenses: Disable or Modify Tools' sub-technique (T1562.001) [83] mentions unhooking only from its 10th version, released in October 2021. The description is abstract and corresponds to what this research categorized as 'binary restoration' and 'engine disarming'.

Detection of the majority of these techniques is possible, even in real time, with low likelihood of false-positives. Runtime indicators and forensic artifacts have been identified wherever applicable. Code tracing and inspection of call stacks in runtime is highly effective against a significant number of these techniques, although this approach has not yet been widely adopted. If adversaries employ call stack spoofing as an additional layer of evasion to thwart detection it is possible to leverage the CET shadow stack [84] or transition-based code tracing [70] when running in a sandbox or emulator to overcome this challenge.

Following the largest global IT outage in history in July 2024, many took to the public stage advocating the prohibition of endpoint security vendors from deploying kernel-based components, even prompting regulators to weigh in.

Purely user-mode-based protection engines have additional limitations: they lack boot-time protection (and early launch anti-malware – ELAM – support), cannot implement self-protection (hardening) and can't monitor all parts of the OS (protected processes, RPC, etc.).

In conclusion, this research showcases that relying solely on user-mode protection engines for security is inadequate due to their fundamental design flaw – the reliance on the same execution environment that is intended to be protected.

REFERENCES

[1] Microsoft. Antimalware Scan Interface (AMSI). https://learn.microsoft.com/en-us/windows/win32/amsi/antimalware-scan-interface-portal.

[2] Chen, R. So what is a Windows "critical process" anyway? Microsoft Dev Blogs. 16 February 2018. https://devblogs.microsoft.com/oldnewthing/20180216-00/?p=98035.

[3] Microsoft. Enable virtualization-based protection of code integrity. https://learn.microsoft.com/en-us/windows/security/hardware-security/enable-virtualization-based-protection-of-code-integrity.

[4] Apple. Endpoint Security. https://developer.apple.com/documentation/endpointsecurity.

[5] Apple. Network Extension. https://developer.apple.com/documentation/networkextension.

[6] Mr-Un1k0d3r / EDRs. https://github.com/Mr-Un1k0d3r/EDRs.

[7] Villeneuve, N.; Eitzman, R.; Nemes, S.; Dean, T. Significant FormBook Distribution Campaigns Impacting the U.S. and South Korea. Google Cloud. 5 October 2017. https://www.fireeye.com/blog/threat-research/2017/10/formbook-malware-distribution-campaigns.html.

[8] Sánchez, D.; Roger, M.; Segura, J. Hancitor: fileless attack with a DLL copy trick. Malwarebytes. 13 March 2018. https://blog.malwarebytes.com/threat-analysis/2018/03/hancitor-fileless-attack-with-a-copy-trick/.

[9] Klopsch, A. Examining Smokeloader's Anti Hooking technique. Malware and Stuff. 24 May 2020. https://malwareandstuff.com/examining-smokeloaders-anti-hooking-technique/.

[10] Cybereason. Operation CuckooBees: A Winnti Malware Arsenal Deep-Dive. https://www.cybereason.com/blog/operation-cuckoobees-a-winnti-malware-arsenal-deep-dive.

[11] Microsoft. SECTION_OBJECT_POINTERS structure (wdm.h). https://learn.microsoft.com/en-us/windows-hardware/drivers/ddi/wdm/ns-wdm-_section_object_pointers.

[12] Zhang, X. Analysis of a New HawkEye Variant. Fortinet. 18 June 2019. https://www.fortinet.com/blog/threat-research/hawkeye-malware-analysis.

[13] Misgav, O. GandCrab Doppelgänged His Shell? Fortinet. 25 July 2019. https://www.fortinet.com/blog/threat-research/gandcrab-doppelganged-his-shell.

[14] hasherezade. Process Doppelgänging meets Process Hollowing in Osiris dropper. Malwarebytes. 13 August 2018. https://blog.malwarebytes.com/threat-analysis/2018/08/process-doppelganging-meets-process-hollowing_osiris/.

[15] Zargarov, N. Fake Update Utilizes New IDAT Loader To Execute StealC and Lumma Infostealers. Rapid7. 31 August 2023. https://www.rapid7.com/blog/post/2023/08/31/fake-update-utilizes-new-idat-loader-to-execute-stealc-and-lumma-infostealers/.

[16] Onofri, D.; Calvelli, E. HijackLoader Expands Techniques to Improve Defense Evasion. Crowdstrike. 7 February 2024. https://www.crowdstrike.com/en-us/blog/hijackloader-expands-techniques/.

[17] Gretzky, K. Defeating Antivirus Real-time Protection From The Inside. Breakdev. 28 July 2016. https://breakdev.org/defeating-antivirus-real-time-protection-from-the-inside/.

[18] Matrosov, A.; Tang, J. You're Off the Hook: Blinding Security Software. ZeroNights. November 2016. https://files.speakerdeck.com/presentations/739577f45e014bfa80c0eca0895f1ead/You_re_Off_the_Hook.pdf.

[19] Haughom, J.; Dantas, J.; Walter, J. LockBit Ransomware Side-loads Cobalt Strike Beacon with Legitimate VMware Utility. SentinelOne. 27 April 2022. https://www.sentinelone.com/labs/lockbit-ransomware-side-loads-cobalt-strike-beacon-with-legitimate-vmware-utility/.

[20] So. C. Earth Freybug Uses UNAPIMON for Unhooking Critical APIs. Trend Micro. 2 April 2024. https://www.trendmicro.com/en_us/research/24/d/earth-freybug.html.

[21] Demboski, M.; Jaramillo, P.; Parsons, M.; Gallagher, S. Operation Crimson Palace: A Technical Deep Dive. Sophos. 5 June 2024. https://news.sophos.com/en-us/2024/06/05/operation-crimson-palace-a-technical-deep-dive/.

[22] Red Team Notes. Full DLL Unhooking with C++. https://www.ired.team/offensive-security/defense-evasion/how-to-unhook-a-dll-using-c++.

[23] Recorded Future. BlueBravo Uses Ambassador Lure to Deploy GraphicalNeutrino Malware. https://go.recordedfuture.com/hubfs/reports/cta-2023-0127.pdf.

[24] Bermejo, L.; Lee, T.; Chen, T. The Espionage Toolkit of Earth Alux: A Closer Look at its Advanced Techniques. Trend Micro. 31 March 2025. https://www.trendmicro.com/en_us/research/25/c/the-espionage-toolkit-of-earth-alux.html.

[25] rb. Perun's Fart - yet another unhooking method. Sector 7. 21 April 2021. https://blog.sektor7.net/#!res/2021/perunsfart.md.

[26] Kalendarov, I. EDR Evasion: A New Technique Using Hardware Breakpoints – Blindside. Cymulate. 5 March 2025. https://cymulate.com/blog/blindside-a-new-technique-for-edr-evasion-with-hardware-breakpoints.

[27] Broumels, T.; Ubink, S. Antivirus evasion by user mode unhooking on Windows 10. https://rp.os3.nl/2020-2021/p68/report.pdf.

[28] Misgav, O. Bypassing User-mode Hooks 101 (BSidesTLV 2019). YouTube. https://www.youtube.com/watch?v=Zk_8nQJeOQg&pp=ygUJYnNpZGVzdGx2&t=1080s.

[29] Vella, C. In-Memory Disassembly for EDR/AV Unhooking. Signal Labs. 2 April 2023. https://signal-labs.com/analysis-of-edr-hooks-bypasses-amp-our-rust-sample/.

[30] Di Cristofaro, D.; Bernardinetti, G. Whisper2Shout – Unhooking technique. SECFORCE. 25 June 2021. https://www.secforce.com/blog/whisper2shout-unhooking-technique/.

[31] Case, A.; Sellers, A.; McDonald, D.; Moreira, G.; Golden G. Richard III. Defeating EDR Evading Malware with Memory Forensics. Volexity. August 2024. https://www.volexity.com/wp-content/uploads/2024/08/Defcon24_EDR_Evasion_Detection_White-Paper_Andrew-Case.pdf.

[32] Evil Socket. On Windows syscall mechanism and syscall numbers extraction methods. 11 February 2014. https://web.archive.org/web/20141002214644/http://www.evilsocket.net/2014/02/11/on-windows-syscall-mechanism-and-syscall-numbers-extraction-methods/.

[33] am0nsec / HellsGate. https://github.com/am0nsec/HellsGate.

[34] Malwarebytes. Floki Bot and the stealthy dropper. 10 November 2016. https://www.malwarebytes.com/blog/news/2016/11/floki-bot-and-the-stealthy-dropper.

[35] Hadar, A. GlobeImposter Ransomware. enSilo. 16 January 2018. https://web.archive.org/web/20190407064851/https://blog.ensilo.com/globeimposter-ransomware-technical.

[36] Gavriel, H. New LockPoS Malware Injection Technique. Cyberbit. 3 January 2018. https://www.cyberbit.com/endpoint-security/new-lockpos-malware-injection-technique/.

[37] Gavriel, H. Latest Trickbot Variant has New Tricks Up Its Sleeve. Cyberbit. 14 August 2018. https://www.cyberbit.com/endpoint-security/latest-trickbot-variant-has-new-tricks-up-its-sleeve/.

[38] Zeligson, A.; Kerner, R. Enter The DarkGate - New Cryptocurrency Mining and Ransomware Campaign. enSilo. 13 November 2018. https://www.fortinet.com/blog/threat-research/enter-the-darkgate-new-cryptocurrency-mining-and-ransomware-campaign.

[39] CyberCoding. Union Api – Call Native Api using syscall. 1 December 2012. https://cybercoding.wordpress.com/2012/12/01/union-api/.

[40] Certego. Malware Tales: Gootkit. 14 February 2019. https://www.certego.net/blog/blog-gootkit/.

[41] Osipov, A. Parallax: The New RAT on the Block. Morphisec. 18 March 2020. https://www.morphisec.com/blog/parallax-rat-active-status/.

[42] Group-IB. The old way: BabLock, new ransomware quietly cruising around Europe, Middle East, and Asia. 4 April 2023. https://www.group-ib.com/blog/bablock-ransomware/.

[43] Vinopal, J.; Yarizadeh, D.; Gekker, G. Rorschach – A New Sophisticated and Fast Ransomware. Check Point Research. 4 April 2023. https://research.checkpoint.com/2023/rorschach-a-new-sophisticated-and-fast-ransomware/.

[44] James. LummaC2 Revisited: What's Making this Stealer Stealthier and More Lethal. SpyCloud. 19 December 2024. https://spycloud.com/blog/lummac2-malware-stealthier-capabilities/.

[45] Haughom, J. From the Front Lines | Hive Ransomware Deploys Novel IPfuscation Technique To Avoid Detection. SentinelOne. 29 March 2022. https://www.sentinelone.com/blog/hive-ransomware-deploys-novel-ipfuscation-technique/.

[46] crummie5 / FreshyCalls. https://github.com/crummie5/FreshyCalls.

[47] jthuraisamy / SysWhispers2. https://github.com/jthuraisamy/SysWhispers2.

[48] rb. Halo's Gate - twin sister of Hell's Gate. Sektor 7. 23 April 2021. https://blog.sektor7.net/#!res/2021/halosgate.md.

[49] trickster0 / TartarusGate. https://github.com/trickster0/TartarusGate.

[50] roy g biv. Heaven's Gate: 64-bit code in 32-bit file. VX Heaven. June 2009. https://web.archive.org/web/20130813132850/http://vxheaven.org/lib/vrg02.html.

[51] Nicolaou, G. Knockin' on Heaven's Gate – Dynamic Processor Mode Switching. RCE. 19 September 2012. https://web.archive.org/web/20140305155654/http://rce.co/knockin-on-heavens-gate-dynamic-processor-mode-switching/.

[52] Microsoft. WOW64 Implementation Details. https://learn.microsoft.com/en-us/windows/win32/winprog64/wow64-implementation-details.

[53] Raashid Bhat Malware Research Blog. Notes On Vawtrak Banking Malware. 18 April 2015. https://int0xcc.svbtle.com/notes-on-vawtrak-banking-malware.

[54] Internet Initiative Japan. Internet Infrastructure Review March 2017 Vol. 34. https://www.iij.ad.jp/en/dev/iir/pdf/iir_vol34_EN.pdf.

[55] Eschweiler, S. YANT-Yet Another Nymaim Talk. Botconf 2017. https://www.botconf.eu/2017/yant-yet-another-nymaim-talk/.

[56] hasherezade. A coin miner with a "Heaven's Gate". 17 January 2018. Malwarebytes. https://www.malwarebytes.com/blog/news/2018/01/a-coin-miner-with-a-heavens-gate.

[57] Nagy, L. Exploting Emotet, an elaborative everyday enigma. Virus Bulletin. 2019. https://www.virusbulletin.com/uploads/pdf/magazine/2019/VB2019-Nagy.pdf.

[58] Cohen, D. A Pony Hidden in Your Secret Garden. CyberArk. 9 May 2019. https://www.cyberark.com/threat-research-blog/a-pony-hidden-in-your-secret-garden/.

[59] Unterbrink, H. RATs and stealers rush through "Heaven's Gate" with new loader. Cisco Talos. 1 July 2019. https://blog.talosintelligence.com/rats-and-stealers-rush-through-heavens/.

[60] Wanve, U. GuLoader: Peering Into a Shellcode-based Downloader. Crowdstrike. 25 June 2025. https://www.crowdstrike.com/en-us/blog/guloader-malware-analysis/.

[61] Misgav, O. Bypassing User-mode Hooks 101. BSidesTLV. YouTube. https://www.youtube.com/watch?v=Zk_8nQJeOQg&pp=ygUJYnNpZGVzdGx2&t=855s.

[62] Secrary.com. Bypass User-Mode Hooks.18 February 2018. https://secrary.com/posts/bypassuserhooks/.

[63] j00ru / windows-syscalls. https://github.com/j00ru/windows-syscalls.

[64] jthuraisamy / SysWhispers. https://github.com/jthuraisamy/SysWhispers.

[65] CyCraft. Chimera APT Threat Report. https://uploads-ssl.webflow.com/6667e1c7aa0aa53cf61a022c/66bc65e430aa86747891a088_%5BTLP-White%5D20200415%20Chimera_V4.2.pdf.

[66] rad9800 / TamperingSyscalls. https://github.com/rad9800/TamperingSyscalls/blob/stripped/TamperingSyscalls/entry.cpp.

[67] Bacurio, F.; Low, W. FortiGuard Labs Threat Research Prevalent Threats Targeting Cuckoo Sandbox Detection and Our Mitigation. Fortinet. 3 January 2018. https://www.fortinet.com/blog/threat-research/prevalent-threats-targeting-cuckoo-sandbox-detection-and-our-mitigation.

[68] Radware. CodeFork Malware Attack. 2017. http://web.archive.org/web/20180628084559/https://security.radware.com/malware/codefork-malware/.

[69] MDSec. FireWalker: A New Approach to Generically Bypass User-Space EDR Hooking. August 2020. https://www.mdsec.co.uk/2020/08/firewalker-a-new-approach-to-generically-bypass-user-space-edr-hooking/.

[70] VMRay. Just Carry A Ladder: – Why Your EDR Let Pikabot Jump Through. 21 March 2024. https://www.vmray.com/just-carry-a-ladder-why-your-edr-let-pikabot-jump-through/.

[71] Dominguez, F. LevelBlue Labs Discovers Highly Evasive, New Loader Targeting Chinese Organizations. LevelBlue. 19 June 2024. https://levelblue.com/blogs/labs-research/highly-evasive-squidloader-targets-chinese-organizations.

[72] CWE. CWE-367: Time-of-check Time-of-use (TOCTOU) Race Condition. https://cwe.mitre.org/data/definitions/367.html.

[73] rad9800 / TamperingSyscalls. https://github.com/rad9800/TamperingSyscalls.

[74] Hutchins, M. Silly EDR Bypasses and Where To Find Them. Malwaretech. 27 December 2023. https://malwaretech.com/2023/12/silly-edr-bypasses-and-where-to-find-them.html.

[75] MalwareJake. Beating up on poor antivirus... 26 July 2013. https://malwarejake.blogspot.com/2013/07/interesting-malware-defense.html.

[76] winternl MemFuck: Bypassing User-Mode Hooks. 8 September 2020. https://winternl.com/memfuck/.

[77] Chester, A. Protecting Your Malware with blockdlls and ACG. XPN's InfoSec Blog. 4 November 2019. https://blog.xpnsec.com/protecting-your-malware/.

[78] Lewis, T.M.; Pimal, B.P. Effects of Removing User-Land Hooks in Endpoint Protection During Attack Experiments. IEEE Access. 23 January 2024. https://ieeexplore.ieee.org/stamp/stamp.jsp?arnumber=10412066.

[79] CCob / SharpBlock. https://github.com/CCob/SharpBlock.

[80] Hutchins, M. Bypassing EDRs With EDR-Preloading. Malwaretech. 13 February 2024. https://malwaretech.com/2024/02/bypassing-edrs-with-edr-preload.html.

[81] van de Wouw, D. Introducing Early Cascade Injection: From Windows Process Creation to Stealthy Injection. Outflank. 15 October 2024. https://www.outflank.nl/blog/2024/10/15/introducing-early-cascade-injection-from-windows-process-creation-to-stealthy-injection/.

[82] Gilboa, A. Evading EDR Detection with Reentrancy Abuse. Deep Instinct. 27 October 2021. https://www.deepinstinct.com/blog/evading-antivirus-detection-with-inline-hooks.

[83] MITRE ATT&CK. Impair Defenses: Disable or Modify Tools. https://attack.mitre.org/versions/v10/techniques/T1562/001/.

[84] Landau, G. Finding Truth in the Shadows. 26 January 2023. Elastic. https://www.elastic.co/security-labs/finding-truth-in-the-shadows.

[85] Misgav, O.; Yavo, U. Bypassing user-mode hooks. FIRST Tel Aviv 2019. https://www.first.org/resources/papers/telaviv2019/Ensilo-Omri-Misgav-Udi-Yavo-Analyzing-Malware-Evasion-Trend-Bypassing-User-Mode-Hooks.pdf.

[86] MDSec. Bypassing User-Mode Hooks and Direct Invocation of System Calls for Red Teams. December 2020. https://www.mdsec.co.uk/2020/12/bypassing-user-mode-hooks-and-direct-invocation-of-system-calls-for-red-teams/.